Function Follows Form: Architecture and 21st Century Software Engineering

Richard N. Taylor and Eric M. Dashofy

Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425, USA
{taylor,edashofy}@ics.uci.edu

ABSTRACT

While the principles and approaches for software development created and advocated by David Parnas and his colleagues have provided primary guidance for engineers for two decades, they are no longer sufficient or appropriate. Products and their contexts as well as the development processes and their contexts have changed dramatically. A new set of principles and approaches are needed. This paper argues for a particular set, chief of which is the use of software architectures as the primary concept in development. We also argue that certain specific architectural styles be adopted as the dominant paradigms, in particular event-based notification and, to a somewhat lesser extent, peer-to-peer architectures. No new technologies or techniques are presented. Rather, this paper is an exploration of priorities for developers, researchers, and educators.[1]

1. This paper is a condensed and edited version of "Moving On: Software Engineering Paradigms for the 21st Century" by Richard N. Taylor appearing in the *Proceedings of the Working Conference on Complex and Dynamic Systems Architectures,* Brisbane, Australia, December 12-14, 2001.

Function Follows Form: Architecture and 21st Century Software Engineering

Richard N. Taylor and Eric M. Dashofy
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425, USA
{taylor,edashofy}@ics.uci.edu

Abstract

While the principles and approaches for software development created and advocated by David Parnas and his colleagues have provided primary guidance for engineers for two decades, they are no longer sufficient or appropriate. Products and their contexts as well as the development processes and their contexts have changed dramatically. A new set of principles and approaches are needed. This paper argues for a particular set, chief of which is the use of software architectures as the primary concept in development. We also argue that certain specific architectural styles be adopted as the dominant paradigms, in particular event-based notification and, to a somewhat lesser extent, peer-to-peer architectures. No new technologies or techniques are presented. Rather, this paper is an exploration of priorities for developers, researchers, and educators.[1]

Keywords: Software development principles, software architectures, event-based notification, peer-to-peer architectures.

1 Introduction

The 2001 International Conference on Software Engineering included a workshop devoted to the examination and recognition of the work of David Parnas [1][2]. Undoubtedly, Parnas's contributions in the areas of modularity, abstract interfaces, program families, and information hiding have, for many years, provided the basis for sound software engineering practice as well as education. As the world around us has changed and as new challenges for software development have appeared; however, it is increasingly clear that these principles and techniques, venerable though they may be, are no longer sufficient.

Several factors exist today that were not present or relevant in Parnas' software engineering climate. Software of the 21st century is characterized by complex, distributed systems-of-systems composed of reused, heterogeneous parts. Software research from the past fifteen years or so demonstrates remarkable capabilities—or at least promise—in addressing issues related to such systems. Drawing from those developments, this paper highlights what we believe to be a superior set of approaches and technologies for guiding development in the early part of the 21st century and briefly discuss the reasons behind these choices.

2 Background: What's Changed?

The contemporary context shows significant differences from that encountered by Parnas in four ways: the nature of the product is different, the context in which a product is used is different, the process by which products are produced is different, and the context that drives the development process is different.

Many of Parnas' writings deal with software products that are "critical": command and control applications, often of a military nature. While critical applications are still important today, modern software products range from tiny embedded applications dedicated solely to displaying the temperature in a living room to games to customer relationship management (CRM). Parnas's products were often created from scratch. Today, applications are seldom created afresh, and most often must fit within a large information systems context.

This context for applications is continually expanding and changing. A new CRM system must work with Internet protocols, employ standard security and encryption technology, reside atop multiple commodity desktop platforms, utilize XML-based documents, incorporate legacy databases, export selected functionality to PDAs, and not infringe a competitor's related patents. Writing a program does not appear nearly as difficult as interoperating with a complex web of other software systems.

This expanding context has created an incredibly rich infrastructure on which to base new systems, and rich and powerful tools and frameworks for assisting in application development. Networking support, distributed middleware and user interface toolkits all represent infrastructure elements that are not developed, but used. In contrast, Parnas's approach to the A-7 OFP involved inventing new technologies at each step, including nascent steps toward creating yet another programming language.

The process by which software is developed is similarly different. For instance, while lip service is still paid to the "need" for abstract requirements specifications, the reality of schedules, the frequent absence of a single, specific customer, as well as the reality of how people design has rendered specs to the "we'll get to it sometime" category.

1. This paper is a condensed and edited version of "Moving On: Software Engineering Paradigms for the 21st Century" by Richard N. Taylor appearing in the *Proceedings of the Working Conference on Complex and Dynamic Systems Architectures*, Brisbane, Australia, December 12-14, 2001.

Consequently a variety of new development processes have emerged that either ignore requirements specs or at least sharply downplay their role. Though decried by the traditionalists in the academic community, these processes reflect the need to stay in business, to establish marketplaces, and to negotiate effectively with users.

One reason for such radical changes in process is the recognition that some of the qualities which Parnas (and most software engineering textbooks) took as sacrosanct are no longer paramount (if they ever were). Chief among these is "correctness". In all but the most critical of applications, the value of correctness may be sharply lower than that of utility, or usability, or robustness, or performance. Systems that have flaws but which work most of the time are often far more valuable than "correct" applications that are delivered too late to satisfy a business need. The situation now is that the critical qualities that a systems needs to exhibit vary — across customers, development organizations, and changing usage contexts. Contemporary development processes must account for such variance.

In Parnas' day, many software systems were developed in the context of a well-understood (and, thus, well-specified) engineering problem. Working in the absence of a classically bounded engineering problem is one of the key challenges for many contemporary developers. Development of standard requirements specifications may now be contemporaneous with development of solution architectures. Parnas, to some degree, anticipated and adopted this position in his paper, "A Rational Design Process and How to Fake It" [3]. Some are explored by Bashar Nuseibeh in [5]. Adding to this, changes in the business context of software development have occurred: an organization may only be willing to engage in a development project if it is convinced that the solution architecture closely enough resembles the architectures of its previous products that an aggressive development and pricing schedule is achievable. Alternatively, there may not even be a specific "customer" identified, and the developers can modify the feature list and user interface as development proceeds, iteratively tuning the product to the needs of the open market. In still other situations we may not be able (or willing, or desirous) to effectively characterize the desired system in terms of abstract qualities; rather solution architectures provide the most effective, convincing communication vehicle between customers and developers.

3 "New" Structuring Paradigms

These issues of context, process, and priorities are clearly critical and have numerous consequences for contemporary developers. We use these concerns and perspectives in choosing the technical solution techniques for software design that are the focus of the remainder of this paper.

3.1 Architectures as the dominant solution

In "Designing Software for Ease of Extension and Contraction" Parnas introduced the notions of program families: applications which have many modules in common, as well as common structure. Parnas's notions of families were strictly focused on subset capabilities, however, and how unwanted dependencies between modules could be eliminated.

Software architecture research, which significantly expands on these ideas, emerged a few years later, and has since blossomed into what we believe should be the key technological focus for software development in the coming years.

Software architecture in this context is not referring to vague "box and arrow" diagrams. Rather, it refers to precise notions of components, connectors, and their interconnection topology along with supporting tools and techniques for creating and evolving running applications. That is, it is not merely a technique for designing or documenting a system, but rather a set of formalisms, design approaches, and tools that are faithful to and exist alongside real software systems. (For a description of key concepts in software architectures, see [4][6][7].) Along with the basics of software architectures, architectural *styles* capture common solution approaches — often properties of topologies — useful in a wide category of applications. Classical notions of objects, classes, and gang-of-four patterns exist at a lower level of design.

We briefly list here seven characteristics of software architecture that argue for its primacy in modern development.

First, software architecture represents a level of abstraction significantly higher than source code, simultaneously effective for reasoning about application structure as well as mapping to large-grain customer issues.

Second, architectural styles and domain-specific software architectures are effective conveyors of knowledge about an application domain, and can thus serve as definers of market spaces and niches [10].

Third, the level of abstraction represented by software architectures is appropriate for reasoning about the relationship between software structure and complex inter-organizational relationships.

Fourth, architectures appear to be the effective level for addressing reuse issues, including the incorporation of black-box components.

Fifth, architectures can provide an effective technical basis for supporting application dynamism—that is, evolving or adapting a running application on-the-fly via modifications to the architecture.

Sixth, architectures represent an effective level of abstraction for dealing with heterogeneity and distribution. As such they represent the level of abstraction appropriate for deciding issues related to middleware and various infrastructural issues.

Seventh, architectures provide an effective basis for supporting configuration management and software deployment concerns, relating those issues to software structure [8][9].

3.2 Key Architectural Styles

While architectural concepts represent the essential, central focus for development, two specific architectural notions are so significant that they deserve attention as possible dominant paradigms for the coming years. First of these is event-based integration, an architectural style that supports loosely-coupled, distributed and dynamic architectures. Second are peer-to-peer architectures, which show great promise for building robust, flexible applications in emerging domains.

3.2.1 Event-based Notification

Event-based notification is an enabling technology for loosely-coupled, highly dynamic, distributed systems. It is also a critical enabler for creating opportunistic and reactive applications—systems which listen for changes in the changing world around them and act in response. The core ideas of event-based notification, or EBN, were recognized twenty or more years ago in the research community, but have been only recently extended and sharpened to make EBN an essential element of emerging Internet applications [12]. Often called “publish-subscribe,” in its most simple form, a software component is notified whenever some particular “event” occurs. The recipient of the notification may choose to take some action in response or perhaps to ignore it; the EBN concept does not require any consequence to receipt.

Given the simplicity of the idea and its generality, it is no wonder that the concept has seen wide use and that many applications could legitimately claim to be using it in some way. The recent technical advances in EBN have come in creating general and scalable mechanisms that allow this simple concept to be employed in very sophisticated and powerful ways.

EBN’s loose coupling of components mitigates some of the incidental aspects of software construction, such as technological domain boundaries: programming languages, hardware platforms, and operating systems processes. This loose coupling can help to surmount non-technological boundaries, for instance, boundaries in organizational structure. While agreement on APIs can help integration across these divides, they can be very fragile. Continual re-negotiation of interfaces is costly and time-consuming. EBN, with reduced requirements on consensus, can lessen the difficulties.

Loose coupling, as enabled by EBN, is a key facet of supporting architectural dynamism—that is, changes to a (distributed) software architecture that occur while it is running. We believe that highly adaptable software systems that can reconfigure themselves dynamically in reaction to internal or external events will become increasingly important in the near future, and that EBN architectures will play an important role in the development of such systems.

Event-based notification can also enable highly distributed, opportunistic, reactive systems. EBN-based architectures are inherently reactive; events proceed from the place of their occurrence to many “interested parties” which initiate action based upon the event. Reactive systems are often designed based on some *a priori* knowledge of what type of events are of interest and what specific action(s) to trigger in response to receipt of a relevant event. However, designers of information systems may not be able to predict in advance all the kinds of information—represented by event notifications—they may be able to utilize. New information sources may emerge, providing new event content. Opportunistic EBN systems proceed a significant step further, searching for and subscribing to classes of information from new sources.

The flip side of opportunistic integration is becoming information infrastructure for others. That is, other applications may be able to opportunistically take advantage of information you possess. EBN systems provide the way for you to publish your information without necessarily knowing in advance the ways or the systems that might utilize the information.

3.2.2 Peer-to-peer (p2p) architectures

Just as event-based notification represents a particular and valuable style of software architecture, so too does peer-to-peer. The essence of p2p architectures is that peers are independent agents capable of performing useful work entirely on their own, and which may derive additional abilities through cooperation with other peers. The distinguishing feature of such applications is that a participant in a peer-to-peer (p2p) network obtains or provides information to other peers directly, without the use of a central or administrative intermediary.

The key technology that enables any peer-to-peer system to work is the use of a defined protocol to govern the interactions between peers. As long as all peers agree to abide by the same protocol, the peers may be implemented in different programming languages, depend on different operating system functionality, behave with different performance characteristics, offer different interfaces to end users, and offer different “added value” functionality to those users. This provides a measure of loose coupling similar to that found in EBN architectures. Notably, one of the key services provided by many p2p protocols is event-based notification. A wide variety of p2p systems and infrastructure are available, with more emerging frequently.

As a design approach it thus places a primacy on intrinsic support for distribution, heterogeneity, and mobility. In a p2p application all the platforms involved (the peers) may not be under a common administrative authority. Indeed, in many popular p2p applications the administrative authority could best be described as anarchic. Consequently p2p architects must consider from the outset how issues of trust, security, unreliability, failure, and non-repeatability are handled. The discipline of developing systems with these properties thus yields significant benefits over the long-haul.

The benefit of addressing these issues is new opportunity. Many p2p systems are remarkably robust. Since one peer cannot depend on the existence of any central server, a functioning regime can potentially be established which, in the aggregate, is able to tolerate failure (or Byzantine action) by other peers.

Because administration is not centralized, new opportunities exist to create systems that interact across organizational and administrative boundaries. The p2p style thus realizes in software the type of relationships we see in teams, whether between individual people, small work groups, companies, or nations. This is clearly a narrower scope than event-based integration, yet obviously it has enormous potential applicability.

In short, as a design discipline, since p2p places a primacy on independence, decentralization, and adaptability, it nicely maps to many contemporary application situations and thus provides a key structuring approach. Technology for supporting p2p applications can be found, e.g., in the Magi software [11] and in the Cougaar project [14].

4 Conclusion

Despite the discussion in many software engineering textbooks and at conferences, Parnas' many contributions are no longer sufficient to address the issues encountered in building modern software systems. Proofs of correctness are not the ultimate goal of software engineering. In today's world, products, processes, and their contexts have changed dramatically such that a new set of principles and approaches need to be identified as the leading, primary concepts. We have argued that chief among these concepts is the use of software architectures as the primary concept in development; that requirements be relegated to a much lesser role in the process, and that certain specific architectural styles be adopted as dominant paradigms. Chief among these styles are event-based notification and peer-to-peer architectures. While it appears that much of the software engineering field is now focusing on measuring the effects of traditional practices and tooling support for quality assurance techniques, it seems like gearing up to fight a previous war. In contrast, we should be aggressively fleshing out these new(er) approaches and educating today's students in more forward-looking technologies.

5 Acknowledgments

The authors are particularly indebted to Ken Anderson.

Effort sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-94-C-0195, F30602-97-2-0021, and F30602-99-C-0174. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory or the U.S. Government.

6 References

[1] Proceedings of the 2001 International Conference on Software Engineering, Toronto, Ontario, Canada. May 2001. IEEE Computer Society Press.

[2] Software Fundamentals: Collected Papers by David L. Parnas. Edited by Daniel M. Hoffman and David M. Weiss. Addison-Wesley, 2001.

[3] Parnas, D.L.and Clements, P. C. 1986. A rational design process: How and why to fake it. IEEE Trans. Softw. Eng. SE-12, 2 (Feb.), 251--257.

[4] Perry, D. E. and A. L. Wolf (1992). "Foundations for the Study of Software Architecture." ACM SIGSOFT Software Engineering Notes 17(4): 40-52.

[5] Nuseibeh, B. Weaving Together Requirements and Architecture, IEEE Computer, 34(3):115-117, March 2001

[6] Medvidovic, N. and R. N. Taylor (2000). "A Classification and Comparison Framework for Software Architecture Description Languages." IEEE Transactions on Software Engineering.

[7] Kruchten, P. (1995). The 4+1 View Model of Architecture. IEEE Software. 12: 42-50.

[8] van der Hoek, A., D. M. Heimbigner, et al. (1998). Software Architecture, Configuration Management, and Configurable Distributed Systems: A Ménage a Trois. Boulder, Colorado, University of Colorado at Boulder.

[9] Hall, R. S., D. Heimbigner, et al. (1999). A Cooperative Approach to Support Software Deployment Using the Software Dock. 1999 International Conference on Software Engineering, Los Angeles, CA, IEEE Computer Society.

[10] Fielding, R. (2000). Architectural Styles and the Design of Network-based Software Architectures (Ph.D. Dissertation). Information and Computer Science. Irvine, CA, University of California, Irvine.

[11] http://www.endeavors.com/

[12] Taylor, R. An Introduction to Event-Based Notification and Event-Based Integration: A KnowNow Whitepaper, June 2001. http://www.knownow.com/products/whitepapers/ebi_whitepaper.pdf

[13] The Workshop on Internet-scale Software Technologies: Internet-scale Namespaces. August 19-20, 1999 University of California, Irvine. http://www.ics.uci.edu/IRUS/twist/twist99/

[14] http://www.cougaar.org/